# Part 1: Introduction

Each journey starts with a first step. Our first step will be to learn about what OpenSSL is, take a look at its history, and learn about what's new in OpenSSL 3. We will also review other SSL/TLS libraries that are competitors of OpenSSL.

This part contains the following chapter:

- *Chapter 1, OpenSSL and Other SSL/TLS Libraries*

# 1
# OpenSSL and Other SSL/TLS Libraries

Currently, there are several libraries available to developers who want to add support for cryptography or SSL/TLS features to their applications. This chapter will help you compare different libraries and learn about the strengths of OpenSSL. You will also learn a bit about OpenSSL's history and what's new in OpenSSL 3.0.

In this chapter, we are going to cover the following topics:

- What is OpenSSL?
- The history of OpenSSL
- What's new in OpenSSL 3.0?
- Comparing OpenSSL with GnuTLS
- Comparing OpenSSL with NSS
- Comparing OpenSSL with Botan
- Comparing OpenSSL with lightweight TLS libraries
- Comparing OpenSSL with LibreSSL
- Comparing OpenSSL with BoringSSL

## What is OpenSSL?

**OpenSSL** is an open source software toolkit that includes a **cryptography** and **SSL/TLS** library, as well as command-line utilities that use the library to provide some useful functionality on the command line, such as generating encryption keys and **X.509 certificates**. The main part of OpenSSL is its library, which means that OpenSSL is mainly useful for software developers. However, system administrators and DevOps specialists will also find OpenSSL's command-line utilities very useful.

**SSL** stands for **Secure Sockets Layer**. It is a protocol designed to provide secure communications over insecure computer networks. An insecure computer network means a network where the transmitted data can be read or even changed by a malicious intermediate network node. An example of such an insecure network is the internet. Secure communication is where transmitted data cannot be read or changed. SSL achieves communication security by using **symmetric** and **asymmetric** cryptography. The SSL protocol was invented in 1995 by the *Netscape Communications Corporation* and was deprecated in 2015 in favor of its successor, the TLS protocol. **TLS** stands for **Transport Layer Security**.

The OpenSSL toolkit was created before the SSL protocol became deprecated, so it contains "SSL" instead of "TLS" in its name.

OpenSSL was historically licensed under the *BSD-style license*, but since version 3.0, it is licensed under *Apache 2.0 license*, which is also BSD-style. This license allows OpenSSL to be used in both open source and closed source applications.

OpenSSL supports a lot of **cryptographic algorithms**, among which are algorithms for **symmetric** and **asymmetric encryption**, **digital signatures**, **message digests**, and **key exchange**. OpenSSL supports **X.509 certificates**, **SSL**, **TLS**, and **DTLS** protocols, as well as other cryptography-related technologies that are less popular.

OpenSSL has been around for a while and during its development, it gained support for a lot of operating systems. OpenSSL was originally developed for Unix-like operating systems. Up until now, OpenSSL supports different variations of Unix, including *GNU/Linux*, *BSDs*, and old and new commercial Unixes, such as *IBM AIX* and *macOS*. OpenSSL also supports popular non-Unix operating systems such as *Microsoft Windows*, mobile OSes such as *Android* and *iOS*, and even old and exotic OSes such as MS-DOS and VMS.

Through many years of OpenSSL development, the library has received numerous optimizations, including *assembly optimizations* for the most popular CPU architectures, such as **x86**, **x86_64**, and **ARM**. OpenSSL is currently one of the fastest existing crypto and TLS libraries.

Because of its universality, support for a lot of algorithms and operating systems, and because of its speed, OpenSSL has become the de facto industry standard. OpenSSL is so popular that other TLS libraries make so-called OpenSSL **compatibility layers** so that those libraries can be used via OpenSSL **application programming interfaces** (**APIs**).

OpenSSL is quite a popular library but what did its path to the widespread adoption look like? Let's find out by walking through the OpenSSL history.

## The history of OpenSSL

The OpenSSL library is based on the **SSLeay library** by Eric Andrew Young. The *eay* in SSLeay stands for *Eric Andrew Young*. SSLeay's development started in 1995 as an open source implementation of the SSL library. Back then, the *NSS* library was not available. Later, Tim Hudson joined the development team. But in 1998, both Eric and Tim were hired by the *RSA Corporation* and did not have time to develop SSLeay further.

SSLeay was forked as OpenSSL in 1998, which means that OpenSSL has become SSLeay's successor. The initial founding members were Mark Cox, Ralf Engelschall, Stephen Henson, Ben Laurie, and Paul Sutton. The very first version of OpenSSL, numbered 0.9.1, was released on December 23, 1998, merely a week after Eric and Tim joined the RSA Corporation and effectively stopped working on SSLeay.

Over many years, a lot of people and companies contributed a lot of code and other work to OpenSSL. The list of contributing companies is impressive: Oracle, Siemens, Akamai, Red Hat, IBM, VMware, Intel, and Arm, among others.

Currently, OpenSSL development is managed by the OpenSSL Management Committee, which consists of seven members. The core development team, which has commit rights, consists of approximately 20 people. Only two people work full-time on OpenSSL. The other contributors either do so in their spare time or as a part of their work in the contributing companies.

The history of OpenSSL is interesting, but what does the future hold for it? Let's find out about the latest changes in the toolkit.

## What's new in OpenSSL 3.0?

One major change in **OpenSSL 3.0** is its license. A software project does not change its license very often during its lifetime. The OpenSSL project used its *BSD-style* open source license until version 3.0. Since version 3.0, it uses *Apache License 2.0*.

OpenSSL 3.0 is a release with big changes in the internal architecture of the library. The architectural changes are not finished and will be continued in OpenSSL 4.0. The concept of **OpenSSL operation implementation providers** was introduced. A **provider** is a unit of code that provides the implementation of cryptographic algorithms. The existing OpenSSL cryptography code will mostly be available via *Default* and *Legacy* providers. **Engines** are still supported in OpenSSL 3.0 but have been deprecated in favor of providers. Support for **ENGINE API** may be removed in OpenSSL 4.0. There is also support for third-party providers that allow independent developers to plug their cryptographic algorithms into OpenSSL.

Another interesting feature of OpenSSL 3.0 is **Kernel TLS** (**KTLS**). When using KTLS, an application can create a special **TLS socket**, similar to a TCP socket. OpenSSL then performs a **TLS handshake** and hands the negotiated encryption key and other data to the **operating system kernel** in the form of **TLS socket options**. Then, the actual data transmission in the TLS protocol is handled by the KTLS code. Such TLS offloading to the kernel can speed up data transmission on high-load systems where performance is important, especially when the kernel can use hardware acceleration for **Advanced Encryption Standard** (**AES**) and thus offload the main CPU. Of course, KTLS support is needed both in the TLS library and in the operating system kernel. At the time of writing, only the Linux and FreeBSD kernels support KTLS.

Some other notable changes in OpenSSL 3.0 include the following:

- Support for the **Certificate Management Protocol**.
- A simple HTTP/HTTPS client.
- **Advanced Encryption Standard Galois/Counter Mode with Synthetic Initialization Vector** (**AES-GCM-SIV**) encryption.
- New **Message Authentication Code** (**MAC**) algorithms, such as **GMAC** and **KMAC**.
- New **Key Derivation Function** (**KDF**) algorithms, such as **SSKDF** and **SSHKDF**.
- New high-level APIs, such as **EVP_MAC**, **EVP_KDF**, and **EVP_RAND**.
- Low-level APIs deprecated in favor of newer higher-level APIs.
- Code cleanup.
- Error handling reworked.
- Old insecure algorithms are no longer available at the default security level.
- Interactive mode has been removed from the `openssl` command-line tool.

OpenSSL is a solid mature software toolkit, so the most important features are already implemented in it. As a result, the latest changes don't contain that much new functionality for a lot of users. The latest release focuses on architectural improvements to the library.

While OpenSSL is the most popular crypto/TLS library, it's not the only one. We'll compare OpenSSL to its competitors in the following sections.

## Comparing OpenSSL with GnuTLS

**GnuTLS** is a free software TLS library that was created for the needs of the **GNU Project**. When GnuTLS was created, most applications of the GNU Project were distributed under the *GPL 2.0 license*, which is incompatible with the old OpenSSL license. The authors of the GPL 2.0 licensed software had to include a licensing exception if they wished to link with OpenSSL. GnuTLS was originally licensed under *LGPL 2.0*, so it did not require such licensing exceptions.

Currently, GnuTLS is licensed under *LGPL 2.1*. This license allows users to use the library in **free and open source software** (**FOSS**) projects. It is also allowed to use the library in closed source projects, but with certain conditions, such as only **dynamic linking**.

GnuTLS does not include cryptography, big-number arithmetic functionality, and some other functionality that OpenSSL includes. Instead, GnuTLS *uses other libraries* from the **GNU ecosystem** that provide the needed functionality: **Nettle** for cryptography, **GMP** for big-number arithmetic, **Libtasn1** for **ASN.1** (short for **Abstract Syntax Notation One**), and so on.

An interesting feature of GnuTLS is that it supports not only X.509 certificates but also OpenPGP certificates. Unlike an X.509 certificate, which is signed by its issuer at the time of being issued, an OpenPGP certificate supports the so-called *web of trust*, can have multiple signatures, and signatures can be added once the certificate has been issued. Unfortunately, OpenPGP certificates have not gained popularity for usage in **TLS connections**.

Apart from OpenPGP certificate support, GnuTLS and its crypto library, Nettle, support fewer crypto algorithms than OpenSSL, and performance-wise, they are a bit slower than OpenSSL. GnuTLS and Nettle, however, support all popular algorithms.

Should you choose OpenSSL or GnuTLS? I recommend that you choose GnuTLS if you are developing GPL-licensed software; otherwise, choose OpenSSL.

Another competitor to OpenSSL is the NSS library, which we will discover in the next section.

## Comparing OpenSSL with NSS

**Network Security Services** (**NSS**) is the first SSL/TLS library. It originated as security code inside the *Netscape Navigator 1.0* browser, released in 1994 by Netscape Communications Corporation. Netscape also invented the SSL protocol, the predecessor of the TLS protocol. Netscape Navigator was succeeded by Netscape Communicator, which was succeeded by the Mozilla browser, which was finally succeeded by Firefox. Soon after the release of Netscape Communicator 4.0 in 1997, the Netscape security code was released as a separate library, named **Hard Core Library** (**HCL**). HCL was later renamed NSS.

The NSS library is licensed under *Mozilla Public License 2.0*, which allows application developers to use the library in both open source and closed source applications.

Notable applications that use the NSS library are applications of Mozilla Foundation, such as Firefox and Thunderbird, some applications from Oracle and Red Hat, and some open source office applications such as LibreOffice and Evolution. While this is a list of solid and respected companies and applications, despite NSS having a long history, it did not become as popular as OpenSSL. I am unsure why. Some people say that OpenSSL is easier to use and has better documentation.

Should you choose NSS or OpenSSL? I recommend OpenSSL because it has better documentation and a larger community of developers and users.

Both NSS and GnuTLS provide their main API in **C**. However, the **Botan** library is different, as we'll find out in the next section.

## Comparing OpenSSL with Botan

Most TLS libraries are written in **C** and provide their main API in C. Botan is a TLS library written in **C++11** that adopted **C++17** in version 3.0. Botan provides its main API in **C++** but also provides **API bindings** for **C** and **Python**. Third-party projects provide API bindings for Ruby, Rust, and Haskell. There are also experimental API bindings for Java and Ocaml.

It's also worth mentioning that the Botan library has good documentation. Botan is distributed under a simple **two-clause BSD license**, which allows users to use the library in both open source and closed source applications.

I recommend Botan for developers who want to use the C++ API and are willing to accept a less popular library with a smaller developer community. If you want to use the C API or want a more performant library with a larger developer community, then stick with OpenSSL.

The Botan library has a larger footprint on your storage than OpenSSL, unlike lightweight TLS libraries, which we will discuss next.

## Comparing OpenSSL with lightweight TLS libraries

Some **lightweight TLS libraries** are available. They are targeted at embedded markets such as **Internet of Things** (**IoT**) devices or other small devices that many people use but they aren't considered computers. This includes payment terminals at your local grocery store, smartwatches, smart light bulbs, and industrial sensors of different kinds. Such devices usually have *weak CPUs*, *low memory*, and *small storage space*.

Lightweight TLS libraries usually become lightweight through *modularity*, the possibility to compile only the needed modules, support fewer cryptographic algorithms and protocols, and have less advanced APIs; that is, they expose fewer features and settings to application developers using the library.

The most well-known lightweight TLS libraries are **wolfSSL** (formerly yaSSL, also known as Yet Another SSL), **Mbed TLS** (formerly PolarSSL), and **MatrixSSL**.

The most advanced of the three libraries seems to be wolfSSL. It supports a lot of cryptographic algorithms, including new algorithms such as **ChaCha20** and **Poly1305**.

The latest versions of wolfSSL have *assembly optimizations* and are very performant. According to the *benchmarks* on the wolfSSL website, woflSSL's raw encryption performance is often on par and will in some cases even noticeably faster than OpenSSL – about 50% faster. Other information on the internet suggests that older versions of wolfSSL are noticeably slower than OpenSSL. If you are interested in performance, then I suggest that you run benchmarking on the target hardware and with the newest versions of the libraries. This is because both libraries are in constant development and newer versions may contain more optimizations.

Mbed TLS and MatrixSSL support noticeably fewer cryptographic algorithms.

WolfSSL and MatrixSSL are dual-licensed under *GPL 2.0* and a commercial license. The GPL 2.0 license only allows users to use the library in GPL-compatible FOSS applications. The commercial license allows users to use the library in *closed source* applications.

Mbed TLS is licensed under *Apache License 2.0*, which allows users to use the library both in open source and closed source applications.

While the lightweight TLS libraries claim to be significantly smaller, approximately 20 times smaller than OpenSSL, they are not so small in their default configurations. Here are the sizes of the libraries, compiled for an *Ubuntu 22.04 x86_64* machine:

- wolfSSL (`version 5.2.0, libwolfssl.so`): 1,768 KiB
- Mbed TLS (`version 2.28.0, libmbedtls.so + libmbedcrypto.so`): 664 KiB
- MatrixSSL (`version 4.5.1, libssl_s.a + libcrypt_s.a + libcore_s.a`): 1,772 KiB
- OpenSSL (`version 3.0.2, libssl.so + libcrypto.so`): 5,000 KiB

To cut down on the library's size, the user of the library has to compile it themselves and disable all the modules that they do not need.

OpenSSL also has a *modular design* and allows you to exclude unneeded modules from the compilation. However, this is more difficult than with lightweight libraries. Some OpenSSL modules are hard to exclude because other modules have dependencies on them, even if they shouldn't. And some OpenSSL modules are just very big, particularly the `x509` module, since it contains the code for working with X.509 certificates. Thus, while it is possible to cut down the size of compiled OpenSSL, it is not possible to cut down as much as with the lightweight TLS libraries.

Use a lightweight TLS library if OpenSSL does not fit into your device. Otherwise, use a *full-size* TLS library, such as OpenSSL.

The last two TLS libraries that we will review originated from OpenSSL itself. Let's find out why the pristine OpenSSL was not good enough for some.

## Comparing OpenSSL with LibreSSL

**LibreSSL** is a fork (derived code) of OpenSSL that was created in 2014 by the OpenBSD Project as a response to the infamous Heartbleed vulnerability that was found in OpenSSL. LibreSSL was founded to increase the *security* and *maintainability* of the library by removing old, unpopular, and no longer secure cryptographic algorithms and other features.

**OpenBSD** is a Unix operating system, one of the **BSD** systems family aimed at security. OpenBSD developers do not only develop the operating system kernel and utilities. Another famous software project is **OpenSSH**. Other software projects started as applications for OpenBSD, such as OpenNTPD, OpenSMTPD, and others. This now includes LibreSSL.

After forking, the LibreSSL developers removed a lot of the original OpenSSL code that they considered old or insecure. They claimed that they removed approximately half of the OpenSSL code within the first week. They also added a few new cryptographic algorithms such as **Advanced Encryption Standard in Galois/Counter Mode** (**AES-GCM**) and **ChaCha-Poly1305**. In addition to adding and removing code, some existing code was reworked and hardened in some places.

Despite LibreSSL's aim to increase security, there were some vulnerabilities in LibreSSL that did not affect OpenSSL, such as CVE-2017-8301.

In the last few years, OpenSSL also did some work on improving the security and maintainability of the library and deprecated/removed some code – though not as much code was removed as in LibreSSL. However, more code and new features were added.

OpenSSL has much more development resources than LibreSSL, which means it advances much faster. For example, from the end of 2018 until the beginning of 2021, LibreSSL has merged approximately *1,500 patches from 36 developers*. During the same time, OpenSSL has merged more than *5,000 patches from 276 developers*. Apart from the core OpenSSL development team, OpenSSL receives code contributions from big companies such as Oracle, Red Hat, IBM, and others.

While the focus of LibreSSL is providing a TLS library for the OpenBSD project, it also aims to support other platforms and remain API-compatible with OpenSSL. At the time of writing, the LibreSSL API is compatible with OpenSSL 1.0.1, but does not include all the newest APIs from OpenSSL 1.0.2 and later.

Since its fork from OpenSSL, LibreSSL became the default TLS library on OpenBSD, but on most other operating systems, the adoption has been low. Most Linux distributions continued using OpenSSL, and some decided to try LibreSSL as a system-wide option but decided to drop such support later. Most commercial application vendors that used OpenSSL also decided to stick with it.

In the competition between OpenSSL and LibreSSL, OpenSSL is winning. I recommend that you choose OpenSSL unless you are developing software specifically for the OpenBSD community, in which case you should consider LibreSSL.

Another fork of OpenSSL has been created by the mighty Google corporation, as we'll explore in the next section.

## Comparing OpenSSL with BoringSSL

**BoringSSL** is another fork of OpenSSL that was made public in 2014. BoringSSL was made for the needs of the Google corporation. For years, Google maintained its own patches for OpenSSL for use in various Google products, such as *Chrome*, *Android*, and the server's infrastructure. Finally, they decided to *fork* OpenSSL and maintain their fork as a separate library.

Like LibreSSL, in BoringSSL, Google removed a lot of the original OpenSSL code, which was responsible for supporting old and unpopular algorithms and features. Google also added some functionality that does not exist in OpenSSL. For example, the **CRYPTO_BUFFER** functionality allows you to deduplicate X.509 certificates in memory, thus reducing memory usage. It also allows you to remove OpenSSL's X.509 and ASN.1 code from the application if OpenSSL is linked statically to the application. The X.509 code is a sizeable part of OpenSSL.

Unlike LibreSSL, BoringSSL does not aim for **API compatibility** with OpenSSL, or even with former versions of BoringSSL. Google wants to change the library API at will. It makes sense because Google controls both BoringSSL and the major software projects that use the library, which makes it possible to synchronize the API changes in BoringSSL and those projects. If the API is not kept stable, it is possible to free up development resources that would otherwise be spent on maintaining the old APIs.

But this also means that if someone outside Google wants to use BoringSSL, they should be ready for *breaking changes* in the library API, at the least suitable times. This is very inconvenient for developers who use the library. Google understands this and states that although BoringSSL is an open source project, it is not for general use.

My opinion is that BoringSSL was made open source mostly for third-party contributors to Google projects, such as Chrome and Android.

I do not recommend using BoringSSL in your applications due to its **API instability**. Furthermore, OpenSSL has more features, better documentation, and a much larger community.

With that, we have reviewed several competitors of OpenSSL. You should now understand the main differences between the popular TLS libraries and which library should be used in which case.

Let's proceed to the summary.

## Summary

In this chapter, we have learned what OpenSSL is and why it's needed. We also walked through the history of OpenSSL and had a glimpse into its future. We reviewed other TLS libraries and competitors to OpenSSL and highlighted their strengths and weaknesses.

You should now have a better understanding of which TLS library you should choose in any given situation. And if the situation is not special, I recommend that you choose OpenSSL since many other people have chosen the same, which implies that OpenSSL is the most popular TLS library and the de facto industry standard.

Now that you know why you should choose OpenSSL, you need to learn how to use it. The next few chapters will be practical ones in that we will learn how to use OpenSSL's powerful features – from symmetric encryption to creating TLS connections with X.509 certificates.